

**DAHLGREN DIVISION
NAVAL SURFACE WARFARE CENTER**

Dahlgren, Virginia 22448-5100



NSWCDD/TR-05/94

**SOFTWARE LIBRARY FOR EXPLICIT UNCERTAINTY
MODELING – PROTOTYPE DEVELOPMENT REPORT**

BY DAVID C. GREGORY

SYSTEMS RESEARCH AND TECHNOLOGY DEPARTMENT

SEPTEMBER 2005

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, search existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 2005	3. REPORT TYPE AND DATES COVERED Final	
4. TITLE AND SUBTITLE Software Library for Explicit Uncertainty Modeling – Prototype Development Report			5. FUNDING NUMBERS -----	
6. AUTHOR(s) David C. Gregory				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Commander Naval Surface Warfare Center Dahlgren Division (Code B55) 17320 Dahlgren Road Dahlgren, VA 22448-5100			8. PERFORMING ORGANIZATION REPORT NUMBER NSWCDD/TR-05/94	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER -----	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE -----	
13. ABSTRACT (Maximum 200 words) <p>An approach for automated uncertainty propagation through deterministic models is developed. The mathematical basis for uncertainty propagation based upon error analysis is derived. A modular, reusable, and universally applicable method of implementing the approach in software is presented, with examples in both Formula Translator (FORTRAN)-90 and C++. Verification techniques are developed and tested. The utility of the approach is demonstrated with an example uncertainty analysis. The example model was taken from the chemical and biological warfare hazard prediction simulation, Vapor, Liquid, and Solid Tracking (VLSTRACK). The methods for extending the approach with additional capability are outlined.</p> <p>The modules presented are applicable to many engineering simulations where uncertainty characterization is desired. Furthermore, the modules can be incorporated into existing codes without redevelopment. Applications include sensitivity analysis, systems architecture studies, and research investment planning.</p>				
14. SUBJECT TERMS Uncertainty Propagation, Software Library, Error Analysis			15. NUMBER OF PAGES 34	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

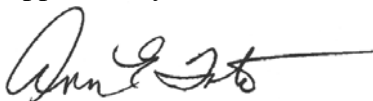
FOREWORD

All engineering and scientific models are affected by the uncertainty of their input parameters. In the case of Chemical and Biological Warfare Hazard Prediction, the models are dominated by input uncertainty. This report develops and demonstrates a technique for treating uncertainty in legacy models without extensive redevelopment. The technique was developed with Hazard Prediction in mind, but it is the author's hope that the technique will be tested and applied to a wide variety of engineering simulations.

The author would like to thank Mr. Max A. Lupton for helping develop the concepts in this report and Mr. Timothy J. Bauer and Mr. Matthew G. Wolski for reviewing the work.

This report has been reviewed by Mr. Gregory J. Sokolowski, Acting Head, Chemical and Biological Defense Battle Management Branch, and Mr. William H. Zimmerman, Acting Head, Chemical and Biological Defense Division.

Approved by:

A handwritten signature in black ink, appearing to read 'Ann E. Tate', with a long horizontal flourish extending to the right.

ANN E. TATE, Head
Systems Research and Technology Department

CONTENTS

	<u>Page</u>
INTRODUCTION	1
APPROACH	1
ANALYTICAL OPERATORS	2
IMPLEMENTATION.....	3
TESTING.....	5
DEMONSTRATION.....	6
RECOMMENDATIONS.....	7
CONCLUSION.....	8
REFERENCES	8
DISTRIBUTION	(1)

APPENDICES

<u>Appendix</u>	<u>Page</u>
A GAUSSIAN DISTRIBUTION OPERATIONS	A-1
B UNCERTAINTY MODULE FORTRAN-90 SOURCE CODE	B-1
C TEST SUBROUTINE FORTRAN-90 SOURCE CODE.....	C-1
D UNCERTAINTY CLASS C++ HEADER FILE.....	D-1
E UNCERTAINTY CLASS C++ SOURCE CODE.....	E-1

TABLES

<u>Table</u>	<u>Page</u>
1 TEST INPUTS	5
2 TESTS AND ANALYTICAL RESULT	5
3 SENSITIVITY OF DROPLET TERMINAL VELOCITY	6

GLOSSARY

Defined-type	In computer programming, a variable whose structure is created by the programmer and consists of variables of intrinsic type, such as integer, real, etc.
Overloading	In computer programming, the process of defining functions or operations of the same name, differing only in the number and/or type of parameters
FORTRAN	Formula Translator
VLSTRACK	Vapor, Liquid, and Solid Tracking (computer model)
VX	O-Ethyl S-Diisopropylaminomethyl Methylphosphonothiolate (chemical nerve agent)

INTRODUCTION

Chemical and biological warfare hazard prediction is dominated by input parameter uncertainty, which has been historically difficult to treat within models. Variations in meteorological conditions, source terms, and even agent properties can have large impacts on the results given by traditional deterministic models. Unfortunately, the traditional approach for uncertainty analyses, the Monte Carlo approach, is computationally expensive even for the simplest of models. The Monte Carlo approach is to run the model once for each data point in a set that represents the statistical uncertainty in the inputs and analyze the output set. Thus, the computational expense is related exponentially to the number of uncertain inputs (O^n). In contrast, the computational expense of the explicit probabilistic approach presented in this technical report is related to the expense of the legacy model by a fixed linear factor.

The purpose of this technical report is to demonstrate an approach for handling uncertainty in existing models explicitly and without resort to Monte Carlo simulations. It is important to note that while the focus of this paper is Chemical and Biological Hazard Prediction, the techniques described are applicable to virtually any scientific or engineering simulation.

APPROACH

Most hazard prediction models, which are traditionally deterministic models, operate upon scalar input. In these models a number represents an input parameter, such as wind speed, with units, for example 5 meters per second. However, if the input parameter has uncertainty associated with it, it would better be represented as a probability distribution. The distribution is a description of the likelihood that the wind speed will be any one scalar value.

The approach developed here is to treat those probability densities in the same ways as the scalar values are handled in the legacy model. If the same functions are derived for operating upon distributions as are used to operate upon the scalar values, then a model can be made to handle uncertainty with little complication. Therefore, functions for addition, multiplication, exponentiation, etc., must be developed for operating upon distribution functions.

Once the operator functions have been derived, features of most modern software languages allow the existing model to be made “uncertain” with very few changes. This approach utilizes two features of modern languages, defined types and overloaded operators. A defined type is a data structure that is defined by the programmer in addition to normal intrinsic types, such as integers, real numbers, booleans, etc. Defined types are created for the various distributions that are used to describe the uncertain input parameters. The power of this approach comes from “overloading” the operators that already exist in the legacy source code,

such as “+”, “*”, “sin()”, with the new functions that have been derived. In this way, the compiler is made to utilize the new operator functions when it encounters an operation involving the new defined types. Thus, existing code can be made uncertainty-enabled by simply changing the variable declarations from the intrinsic types to the new defined types.

The derived operator functions for Gaussian distributions are often algebraic and only slightly more complicated than the scalar operations. Therefore, the numerical complexity of this explicit approach is a multiple of the legacy numerical cost.

It should also be noted that this approach can also be applied to most models that solve integral or differential equations. Integral and differential equations are usually solved numerically using the basic algebraic operations for which the approach outlined in this paper is valid.

ANALYTICAL OPERATORS

The fundamental technology of this explicit approach is development of operators for distribution functions, which have the same behavior as their scalar counterparts. Specifically, the order of operations should not be affected.

For the purposes of the explicit probabilistic prototype, only Gaussian distributions were considered. For distributions where the variance is small when compared to the mean, the error in a function is related to the function’s total differential¹ and the output distribution will also be Gaussian. So, for $f(x,y,z)$, the total differential is written

$$df = \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial y} dy + \frac{\partial f}{\partial z} dz \quad (\text{Equation 1})$$

For Gaussian distributions, the error in f , σ_f , is given by the following equation, where dx is represented by the error in x , σ_x .

$$\sigma_f = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 \cdot \sigma_x^2 + \left(\frac{\partial f}{\partial y}\right)^2 \cdot \sigma_y^2 + \left(\frac{\partial f}{\partial z}\right)^2 \cdot \sigma_z^2} \quad (\text{Equation 2})$$

Most simple operations, such as the addition, subtraction, multiplication, and division of two uncertain values can be readily located in literature² and are summarized in Appendix A. However, many more complicated functions, such as those involving only one uncertain value or transcendental operators, are not available. Some operations not generally published that are currently in the library will be derived here as examples.

Consider the following function:

$$f(x) = \frac{m}{x} \quad (\text{Equation 3})$$

where m is an exact constant and x is a Gaussian distribution given by

$$x = \mu_x \pm \sigma_x \quad (\text{Equation 4})$$

and where μ_x and σ_x are the mean and standard deviation of the distribution. The partial differential of f with respect to x is

$$\frac{\partial f}{\partial x} = \frac{-m}{x^2} \quad (\text{Equation 5})$$

The distribution of f can be shown to be, using Equation 2,

$$f(x) = \frac{m}{x} = \mu_f \pm \sigma_f = \frac{m}{\mu_x} \pm \left(\mu_f \cdot \left(\frac{\sigma_x}{\mu_x} \right) \right) \quad (\text{Equation 6})$$

Next, consider this function, where m is an exact constant and x is a Gaussian distribution:

$$f(x) = m^x \quad (\text{Equation 7})$$

The partial differential of f with respect to x is

$$\frac{\partial f}{\partial x} = m^x \cdot \ln(m) \quad (\text{Equation 8})$$

which can also be written

$$\frac{\partial f}{\partial x} = f(x) \cdot \ln(m) \quad (\text{Equation 9})$$

Therefore, the distribution function can be shown to be

$$f(x) = m^x = \mu_f \pm \sigma_f = m^{\mu_x} \pm (\mu_f \cdot \sigma_x \cdot \ln(m)) \quad (\text{Equation 10})$$

IMPLEMENTATION

As described above, the key concepts for this method are defined types and overloaded operators. The prototype was developed using Formula Translator (FORTRAN)-90 and the description here will follow the source code in Appendix B. However, this approach can be

implemented in other languages. In fact, implementation in C++ or Java is potentially simpler and requires fewer lines than in FORTRAN-90 due to their object-oriented features. A clear illustration of how to implement the approach in C++ is available online.³ A tested and verified skeleton implementation is included in Appendices D and E.

In FORTRAN-90, a defined type is declared as follows:

```
type gaussian
    double precision :: mean
    double precision :: sigma
end type gaussian
```

Note that the type has two members representing the distribution average and standard deviation. The type definition allows variables of type *gaussian* to be declared.

Next, the operations outlined in Appendix A are implemented in source code. A function is written for each operator. In general, the functions will accept two gaussian types as arguments and return a gaussian type. The example shown below is the addition of two gaussian types.

```
type(gaussian) function gaussianAdd (left, right)
    type(gaussian), intent(in) :: left
    type(gaussian), intent(in) :: right

    gaussianAdd%mean = left%mean + right%mean
    gaussianAdd%sigma = SQRT(left%sigma**2 + right%sigma**2)
end function gaussianAdd
```

Lastly, in FORTRAN-90, it is necessary to inform the compiler to overload an operator symbol with the function.

```
interface operator (+)
    module procedure gaussianAdd
end interface
```

These three steps are repeated for each operator and each combination of types. The type declaration, operator functions, and interfaces are collected in a module, which is then easily utilized in other routines as needed. The module source code is available in Appendix B.

Adding uncertainty capability to pre-existing source code is then a simple exercise. Consider the following simple function:

```
real function foo (x,y,w)
    real, intent(in) :: x
    real, intent(in) :: y
    real, intent(in) :: w
    foo = w * y**2 / SQRT(x)
end function foo
```

Modifying the function to use the uncertainty module requires changing only the variable declarations and function header.

```

type(gaussian) function foo (x,y,w)
  type(gaussian), intent(in)      :: x
  type(gaussian), intent(in)      :: y
  type(gaussian), intent(in)      :: w
  foo = w * y**2 / sqrt(x)
end function foo

```

TESTING

Because the uncertainty module utilizes an analytical approach, it must give the exact analytical answer. Thus, the implementation of the approach in the module may be verified by testing each function against the analytical answer generated by hand. A testing subroutine was written that completes seven tests of the functions. The variables in all seven tests are presented in Table 1. The tests and the results are shown in Table 2 and the source code is available in Appendix C.

Table 1. Test Inputs

Variable	Mean	Standard Deviation
w	4.52	0.02
x	2.0	0.2
y	3.0	0.6

Table 2. Tests and Analytical Result

Test ID	Test Function	Analytical Mean	Analytical Standard Deviation
1	$x + y - w$	0.48	0.632771
2	$w * x$	9.04	0.904885
3	x / y	0.666667	0.149071
4	$w ** \text{int}(3)$	92.3454	1.225824
5	$w ** \text{real}(3)$	92.3454	1.225824
6	$w ** \text{dble}(3)$	92.3454	1.225824
7	$w * y**2 / \text{sqrt}(x)$	28.7651	11.59628

Obviously, it would be interesting to compare the analytical-based approach to a Monte Carlo solution. Preliminary investigations have shown that generating an accurate sample from the Gaussian distribution is critical to obtaining the correct analytical result. Results from Monte Carlo runs of a simple multiplication as in Test 2 match the analytical and uncertainty module result. However, the Monte Carlo approach is not able to match the analytical result for Test 7, even with very large numbers of samples. The Gaussian sampling function is suspected to be the source of error and a higher quality sampler is being sought.

DEMONSTRATION

In order to demonstrate the use of the uncertainty module approach, a subroutine derived from the Vapor, Liquid, and Solid Tracking (VLSTRACK) source tree was modified. VLSTRACK is a Chemical/Biological Hazard Prediction software written at the Naval Surface Warfare Center, Dahlgren. VLSTRACK provides an analysis tool for hazards resulting from the deployment of conventional Chemical and Biological Weapons. The function used for this demonstration was taken from the source code of VLSTRACK 3.1.2 and calculates droplet terminal velocity. The return value of the function is the terminal velocity of a liquid droplet given air and agent properties and accounting for nonspherical droplet deformation.⁴

The necessary modification has already been presented in the Implementation section. A simple comparative sensitivity analysis was conducted in order to evaluate the importance of each of the seven input parameters. For each test, the sigma of the variable being tested was set to 10 percent (%) of the nominal value. The sigma for the remaining inputs was set to zero. The relative error in the terminal velocity calculation for the given error in each input parameter is summarized in Table 3. The input parameters are representative of a 500-micron droplet of O-Ethyl S-Diisopropylaminomethyl Methylphosphonothiolate (VX) at sea level.

Table 3. Sensitivity of Droplet Terminal Velocity

Input Parameter	Mean (μ)	Variance ($\sigma = 0.1 * \mu$)	Relative Output Error % (σ / μ)
droplet diameter (m)	0.0005	0.00005	17.0
acceleration of gravity (m/s)	9.801	0.9801	5.15
air viscosity at 20 degrees Celsius (°C) (N*s/m ²)	1.8e-05	1.8e-06	1.78
agent viscosity (N*s/m ²)	0.7	0.07	1.44e-05
agent density (kg/m ³)	1010	101	5.15
air density at 20°C (kg/m ³)	1.2	0.12	5.15
agent surface tension (N/m)	3.2e-02	3.2e-03	0.234

RECOMMENDATIONS

This report serves merely as an introduction to the approach developed. There remains a significant amount of work to extend, validate, and verify the methods. Some potential areas for future work follow in this section.

The Monte Carlo approach should be rigorously compared against the uncertainty module. One potential finding is that the Monte Carlo technique requires very large numbers of samples in order to approximate the analytical solution.

The restriction of the variance being small compared to the mean may be prohibitive for Chemical and Biological Hazard Prediction. There are Hazard Prediction model inputs for which the variance is a significant percentage of the mean value. There are methods for deriving errors under these conditions, involving Taylor series expansions. However, the resultant operations do not typically generate Gaussian distributions. Instead, the Central Limit Theorem says that resulting distributions will be log-normal. Thus, the implementation is more complicated, requiring more defined types and more intensive functions.

The list of operator functions is incomplete. The source code currently contains the simplest operations and a few more complicated ones. A complete list of functions should be sought in connection with further testing with legacy source code.

A complete set of tests should be developed that covers all functions in the uncertainty module. This task is necessary to ensure accurate implementation of the operations.

The execution speed penalty for using the uncertainty module should be quantified and compared to a Monte Carlo approach that yields acceptable accuracy.

Because compilers will provide error messages when variable declarations are changed to the new defined types, modification of existing legacy code can be automated. It would be straightforward to develop an automated conversion utility in a scripting language such as Python.

For universal usage, the uncertainty module should be implemented as a C++ class, as explained by Reference 3. This work has been started and is included as Appendix D and Appendix E.

It will also be necessary to investigate other types of probability distributions. Log-normal and chi-squared distributions are likely to be needed because of the limitations of this approach discussed in the **Analytical Operators** section. A review of existing code should be undertaken in order to determine which distributions are utilized. Then, it will likely be necessary to develop and implement operation functions for mixed distribution operations.

CONCLUSION

This technical report serves to introduce the Explicit Probabilistic Modeling approach. This approach consists of combining properties of Gaussian distributions within a programming framework. The resulting library allows existing software to handle uncertainty calculations without extensive recoding. The approach has been implemented and tested, end-to-end, on a limited scale. Results are presented for an abstract set of test cases as well as for a component of a Chemical/Biological Hazard Prediction model.

Explicit probabilistic modeling has the potential to change the way that Chemical/Biological Hazard Prediction is approached. Because the technique can be applied on a generic, operator basis, pre-existing investments in deterministic models can be retained and made probabilistic. Detailed analyses of the sensitivity of hazard prediction models are possible due to the speed and simplicity of this approach. Extensive design of experimental techniques can be used to identify critical model input parameters. Identifying the most critical input parameters will help focus future research.

Perhaps most importantly, this approach has nearly universal applicability to scientific and engineering simulations. Uncertainty-enabled models would offer many new possibilities, especially in the area of sensitivity studies. Analysis conducted with uncertainty-enabled models would help focus future research and development and empirical model validation.

REFERENCES

1. Bethea, Robert M.; Duran, Benjamin S.; and Boullion, Thomas L., *Statistical Methods for Engineers and Scientists (Second Edition)*, Marcel Dekker, Inc., New York, NY, p. 393.
2. Lindberg, Vern, "Uncertainties and Error Propagation,"
<http://www.rit.edu/~uphysics/uncertainties/Uncertaintiespart2.html>.
3. Microsoft, "C++ Language Reference – Operator Overloading,"
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclang/html/_pluslang_overloaded_operators.asp.
4. Bauer, Timothy J., *Software Design Description for the Chemical/Biological Agent Vapor, Liquid, and Solid Tracking (VLSTRACK) Computer Model, Version 3.0*, NSWCDD/TR-99/6, Naval Surface Warfare Center, Dahlgren Division, Dahlgren, VA.

APPENDIX A
GAUSSIAN DISTRIBUTION OPERATIONS

Convention:

x represents a Gaussian distribution with a mean, μ_x , and standard deviation, σ_x

y represents a Gaussian distribution with a mean, μ_y , and standard deviation, σ_y

m represents a scalar constant or known value with no uncertainty

Operation	Output Mean (μ_w)	Output Standard Deviation
$x + y$	$\mu_x + \mu_y$	$\sqrt{\sigma_x^2 + \sigma_y^2}$
$x + m$	$\mu_x + m$	σ_x
$x - y$	$\mu_x - \mu_y$	$\sqrt{\sigma_x^2 + \sigma_y^2}$
$x - m$	$\mu_x - m$	σ_x
$x * y$	$\mu_x * \mu_y$	$\mu_w \cdot \sqrt{\left(\frac{\sigma_x}{\mu_x}\right)^2 + \left(\frac{\sigma_y}{\mu_y}\right)^2}$
$x * m$	$\mu_x * m$	$\sigma_x \cdot m$
x / y	μ_x / μ_y	$\mu_w \cdot \sqrt{\left(\frac{\sigma_x}{\mu_x}\right)^2 + \left(\frac{\sigma_y}{\mu_y}\right)^2}$
x / m	μ_x / m	$\frac{\sigma_x}{m}$
m / x	M / μ_x	$\mu_w \cdot \frac{\sigma_x}{\mu_x}$
x^m	μ_x^m	$\mu_w \cdot \left m \cdot \frac{\sigma_x}{\mu_x} \right $
m^x	m^{μ_x}	$\mu_x \cdot \sigma_x \cdot \ln(m)$

APPENDIX B

UNCERTAINTY MODULE FORTRAN-90 SOURCE CODE


```

!-----
! NAVSEA Warfare Center - Dahlgren
!-----
!      Created by:      David C. Gregory - 9 August 2005
!
!      Last modified:   dcg - 24 August 2005
!                      dcg - 10 August 2005
!-----

```

module Uncertainty

```

!the root of it all, the gaussian distribution type
type gaussian
    double precision :: mean
    double precision :: sigma
end type gaussian

!... all of the interfaces...
interface compare
    module procedure gaussianCompare
end interface
interface operator (+)
    module procedure gaussianAdd
    module procedure gaussianAddDouble
    module procedure doubleAddGaussian
    module procedure realAddGaussian
    module procedure gaussianAddReal
end interface
interface operator (-)
    module procedure gaussianSubtract
    module procedure gaussianSubtractDouble
    module procedure doubleSubtractGaussian
    module procedure gaussianSubtractReal
    module procedure realSubtractGaussian
    module procedure gaussianCompliment
end interface
interface operator (*)
    module procedure gaussianMultiply
    module procedure gaussianMultiplyDouble
    module procedure doubleMultiplyGaussian
    module procedure gaussianMultiplyReal
    module procedure realMultiplyGaussian
end interface
interface operator (/)
    module procedure gaussianDivide
    module procedure gaussianDivideDouble
    module procedure gaussianDivideReal
    module procedure doubleDivideGaussian
    module procedure realDivideGaussian
end interface
interface operator (**)
    module procedure gaussianPowerDouble
    module procedure gaussianPowerReal
    module procedure gaussianPowerInteger
    module procedure doublePowerGaussian
    module procedure realPowerGaussian
end interface
interface SQRT
    module procedure gaussianSQRT
end interface
interface operator (.lt.)
    module procedure gaussianLess
end interface
interface operator (.le.)
    module procedure gaussianLessEqual
    module procedure gaussianLessEqualDouble
    module procedure gaussianLessEqualReal
end interface
interface min

```

```

    module procedure gaussianMinReal
end interface
interface exp
    module procedure gaussianExp
end interface

! Private functions are not available outside of the module, forcing the operators to be used instead
private :: gaussianCompare

private :: gaussianAdd
private :: gaussianAddDouble
private :: doubleAddGaussian
private :: realAddGaussian
private :: gaussianAddReal

private :: gaussianSubtract
private :: gaussianSubtractDouble
private :: doubleSubtractGaussian
private :: gaussianSubtractReal
private :: realSubtractGaussian

private :: gaussianMultiply
private :: gaussianMultiplyDouble
private :: doubleMultiplyGaussian
private :: gaussianMultiplyReal
private :: realMultiplyGaussian

private :: gaussianDivide
private :: gaussianDivideDouble
private :: gaussianDivideReal
private :: doubleDivideGaussian
private :: realDivideGaussian

private :: gaussianPowerDouble
private :: gaussianPowerReal
private :: gaussianPowerInteger
private :: doublePowerGaussian
private :: realPowerGaussian

private :: gaussianSQRT

private :: gaussianLess
private :: gaussianLessEqual
private :: gaussianLessEqualDouble
private :: gaussianLessEqualReal

private :: gaussianMinReal

private :: gaussianExp

private :: gaussianCompliment

```

contains

!!! The functions for comparing gaussian types

```

logical function gaussianCompare(arg1,arg2,tol)
    type(gaussian), intent(in)      :: arg1
    type(gaussian), intent(in)      :: arg2
    double precision, intent(in)    :: tol

    gaussianCompare = ((abs(arg1%mean - arg2%mean)/arg2%mean) .lt. tol) .and.
    ( (abs(arg1%sigma - arg2%sigma)/arg2%mean) .lt. tol)
end function gaussianCompare

```

!!! All the add operations

```

type(gaussian) function gaussianAdd (left, right)
    type(gaussian), intent(in) :: left

```

```

type(gaussian), intent(in) :: right
gaussianAdd%mean = left%mean + right%mean
gaussianAdd%sigma = SQRT(left%sigma**2 + right%sigma**2)

```

```

end function gaussianAdd

```

```

type(gaussian) function gaussianAddDouble (left, right)
  type(gaussian), intent(in)          :: left
  double precision, intent(in)      :: right

  gaussianAddDouble%mean = left%mean + right
  gaussianAddDouble%sigma = left%sigma

```

```

end function gaussianAddDouble

```

```

type(gaussian) function doubleAddGaussian (left, right)
  type(gaussian), intent(in)          :: right
  double precision, intent(in)      :: left

  doubleAddGaussian%mean = left + right%mean
  doubleAddGaussian%sigma = right%sigma

```

```

end function doubleAddGaussian

```

```

type(gaussian) function realAddGaussian (left, right)
  type(gaussian), intent(in)          :: right
  real, intent(in)                  :: left

  realAddGaussian%mean = dble(left) + right%mean
  realAddGaussian%sigma = right%sigma

```

```

end function realAddGaussian

```

```

type(gaussian) function gaussianAddReal (left, right)
  type(gaussian), intent(in)          :: left
  real, intent(in)                  :: right

  gaussianAddReal%mean = left%mean + right
  gaussianAddReal%sigma = left%sigma

```

```

end function gaussianAddReal

```

!!! All the subtract operations

```

type(gaussian) function gaussianSubtract (left, right)
  type(gaussian), intent(in) :: left
  type(gaussian), intent(in) :: right

  gaussianSubtract%mean = left%mean - right%mean
  gaussianSubtract%sigma = SQRT(left%sigma**2 + right%sigma**2)

```

```

end function gaussianSubtract

```

```

type(gaussian) function gaussianSubtractDouble (left, right)
  type(gaussian), intent(in)          :: left
  double precision, intent(in)      :: right

  gaussianSubtractDouble%mean = left%mean - right
  gaussianSubtractDouble%sigma = left%sigma

```

```

end function gaussianSubtractDouble

```

```

type(gaussian) function gaussianSubtractReal (left, right)
  type(gaussian), intent(in)          :: left
  real, intent(in)                  :: right

  gaussianSubtractReal%mean = left%mean - dble(right)
  gaussianSubtractReal%sigma = left%sigma

```

```

end function gaussianSubtractReal

type(gaussian) function doubleSubtractGaussian (left, right)
    type(gaussian), intent(in)          :: right
    double precision, intent(in)        :: left

    doubleSubtractGaussian%mean = left - right%mean
    doubleSubtractGaussian%sigma = right%sigma

end function doubleSubtractGaussian

type(gaussian) function realSubtractGaussian (left, right)
    type(gaussian), intent(in)          :: right
    real, intent(in)                   :: left

    realSubtractGaussian%mean = dble(left) - right%mean
    realSubtractGaussian%sigma = right%sigma

end function realSubtractGaussian

```

!!! All the multiply operations

```

type(gaussian) function gaussianMultiply (left, right)
    type(gaussian), intent(in) :: left
    type(gaussian), intent(in) :: right

    gaussianMultiply%mean = left%mean * right%mean
    gaussianMultiply%sigma = gaussianMultiply%mean *
        SQRT((left%sigma/left%mean)**2 + (right%sigma/right%mean)**2)

end function gaussianMultiply

type(gaussian) function gaussianMultiplyDouble (left, right)
    type(gaussian), intent(in)          :: left
    double precision, intent(in)        :: right

    gaussianMultiplyDouble%mean = left%mean * right
    gaussianMultiplyDouble%sigma = left%sigma * right

end function gaussianMultiplyDouble

type(gaussian) function doubleMultiplyGaussian (left, right)
    type(gaussian), intent(in)          :: right
    double precision, intent(in)        :: left

    doubleMultiplyGaussian = gaussianMultiplyDouble(right,left)

end function doubleMultiplyGaussian

type(gaussian) function gaussianMultiplyReal (left, right)
    type(gaussian), intent(in)          :: left
    real, intent(in)                   :: right

    gaussianMultiplyReal%mean = left%mean * dble(right)
    gaussianMultiplyReal%sigma = left%sigma * dble(right)

end function gaussianMultiplyReal

type(gaussian) function realMultiplyGaussian (left, right)
    type(gaussian), intent(in)          :: right
    real, intent(in)                   :: left

    realMultiplyGaussian = gaussianMultiplyReal(right,left)

end function realMultiplyGaussian

```

!!! All the divide operations

```

type(gaussian) function gaussianDivide (left, right)
  type(gaussian), intent(in) :: left
  type(gaussian), intent(in) :: right

  gaussianDivide%mean = left%mean / right%mean
  gaussianDivide%sigma = gaussianDivide%mean *
    SQRT((left%sigma / left%mean)**2 + (right%sigma / right%mean)**2) &

end function gaussianDivide

type(gaussian) function gaussianDivideDouble (left, right)
  type(gaussian), intent(in) :: left
  double precision, intent(in) :: right

  gaussianDivideDouble%mean = left%mean / right
  gaussianDivideDouble%sigma = left%sigma / right

end function gaussianDivideDouble

type(gaussian) function gaussianDivideReal (left, right)
  type(gaussian), intent(in) :: left
  real, intent(in) :: right

  gaussianDivideReal%mean = left%mean / dble(right)
  gaussianDivideReal%sigma = left%sigma / dble(right)

end function gaussianDivideReal

type(gaussian) function realDivideGaussian (left, right)
  type(gaussian), intent(in) :: right
  real, intent(in) :: left

  realDivideGaussian%mean = dble(left) / right%mean
  realDivideGaussian%sigma = abs(realDivideGaussian%mean * right%sigma/right%mean)

end function realDivideGaussian

type(gaussian) function doubleDivideGaussian (left, right)
  type(gaussian), intent(in) :: right
  double precision, intent(in) :: left

  doubleDivideGaussian%mean = left / right%mean
  doubleDivideGaussian%sigma = abs(doubleDivideGaussian%mean * right%sigma/right%mean)

end function doubleDivideGaussian

!!! All the power operations

type(gaussian) function gaussianPowerDouble (left, right)
  type(gaussian), intent(in) :: left
  double precision, intent(in) :: right

  gaussianPowerDouble%mean = left%mean ** right
  gaussianPowerDouble%sigma = gaussianPowerDouble%mean *
    abs (right * left%sigma / left%mean) &

end function gaussianPowerDouble

type(gaussian) function gaussianPowerReal (left, right)
  type(gaussian), intent(in) :: left
  real, intent(in) :: right

  gaussianPowerReal%mean = left%mean ** right
  gaussianPowerReal%sigma = gaussianPowerReal%mean *
    abs (right * left%sigma / left%mean) &

end function gaussianPowerReal

type(gaussian) function gaussianPowerInteger (left, right)

```

```

type(gaussian), intent(in)           :: left
integer, intent(in)  :: right

gaussianPowerInteger%mean = left%mean ** right
gaussianPowerInteger%sigma = gaussianPowerInteger%mean *
    abs (right * left%sigma / left%mean)

end function gaussianPowerInteger

type(gaussian) function doublePowerGaussian (left, right)
    type(gaussian), intent(in)           :: right
    double precision, intent(in)  :: left

    doublePowerGaussian%mean = left ** right%mean
    doublePowerGaussian%sigma = abs(right%sigma * doublePowerGaussian%mean * log(left))

end function doublePowerGaussian

type(gaussian) function realPowerGaussian (left, right)
    type(gaussian), intent(in)           :: right
    real, intent(in)  :: left

    realPowerGaussian%mean = dble(left) ** right%mean
    realPowerGaussian%sigma = abs(right%sigma * realPowerGaussian%mean * log(left))

end function realPowerGaussian

type(gaussian) function gaussianExp (arg)
    type(gaussian), intent(in) :: arg

    gaussianExp = doublePowerGaussian(exp(1.0d0),arg)

end function gaussianExp

!!! Square root operation

type(gaussian) function gaussianSQRT (arg)
    type(gaussian), intent(in) :: arg

    gaussianSQRT = gaussianPowerDouble (arg,5.0d-1)

end function gaussianSQRT

!!! Logical Operations
!!!!!! note bene!!!!!!
! These need serious consideration and are to be used only for proof of concept!

logical function gaussianLess (left,right)
    type(gaussian), intent(in) :: left
    type(gaussian), intent(in) :: right

    gaussianLess = left%mean .lt. right%mean

end function gaussianLess

logical function gaussianLessEqual (left,right)
    type(gaussian), intent(in) :: left
    type(gaussian), intent(in) :: right

    gaussianLessEqual = left%mean .le. right%mean

end function gaussianLessEqual

logical function gaussianLessEqualDouble (left,right)
    type(gaussian), intent(in) :: left
    double precision, intent(in) :: right

    gaussianLessEqualDouble = left%mean .le. right

```

```

end function gaussianLessEqualDouble

logical function gaussianLessEqualReal (left,right)
  type(gaussian), intent(in) :: left
  real, intent(in) :: right

  gaussianLessEqualReal = left%mean .le. dble(right)

end function gaussianLessEqualReal

!!! Other intrinsics
!!!! Note Bene !!!!
!!!! these still need serious consideration as well
!           specifically:
!           gaussianMinReal

type(gaussian) function gaussianMinReal (arg1,arg2)
  type(gaussian), intent(in) :: arg1
  real, intent(in) :: arg2

  if (arg1%mean .lt. dble(arg2)) then
    gaussianMinReal%mean = dble(arg2)
    gaussianMinReal%sigma = arg1%sigma
  else
    gaussianMinReal = arg1
  end if

end function gaussianMinReal

type(gaussian) function gaussianCompliment (arg)
  type(gaussian), intent(in) :: arg

  gaussianCompliment%mean = -arg%mean
  gaussianCompliment%sigma = arg%sigma

end function gaussianCompliment

end module Uncertainty

```

APPENDIX C

TEST SUBROUTINE FORTRAN-90 SOURCE CODE


```

integer function testUncertaintyLibrary ()
  double precision :: tol
  integer :: errCount
  type(gaussian) :: x
  type(gaussian) :: y
  type(gaussian) :: z
  type(gaussian) :: w
  type(gaussian) :: answer1,answer2,answer3,answer4
  type(gaussian) :: answer5,answer6,answer7,answer8
  type(gaussian) :: foo7

  !tolerance for this test suite
  tol = 0.00001

  !control variables for all the tests
  w = gaussian(4.52,0.02)
  x = gaussian(2.0,0.2)
  y = gaussian(3.0,0.6)

  !answers for all the tests
  answer1 = gaussian(0.48000,0.632771)
  answer2 = gaussian(9.04000,0.904885)
  answer3 = gaussian(0.66667,0.149071)
  answer4 = gaussian(92.3454,1.225824)
  answer5 = gaussian(92.3454,1.225824)
  answer6 = gaussian(92.3454,1.225824)
  answer7 = gaussian(28.7651,11.59628)

  !Test #1: addition and subtraction
  z = x + y - w
  errCount = errCount + setErr( compare(z,answer1,tol) )

  !Test #2: multiplication
  z = w*x
  errCount = errCount + setErr( compare(z,answer2,tol) )

  !Test #2: division
  z = x/y
  errCount = errCount + setErr( compare(z,answer3,tol) )

  !Test#4: test the double power function
  z = w**(dble(3))
  errCount = errCount + setErr( compare(z,answer4,tol) )

  !Test#5: test the real power function
  z = w**(real(3))
  errCount = errCount + setErr( compare(z,answer5,tol) )

  !Test#6: test the integer power function
  z = w**(int(3))
  errCount = errCount + setErr( compare(z,answer6,tol) )

  !Test7 #: multiplication,division, powers, and squareroots
  z = (w*y**2)/SQRT(x)
  errCount = errCount + setErr( compare(z,answer7,tol) )

  testUncertaintyLibrary = errCount
end function testUncertaintyLibrary

```

APPENDIX D

UNCERTAINTY CLASS C++ HEADER FILE

Gaussian.h

```

//-----
// NAVSEA Warfare Center - Dahlgren
//-----
//      Created by:      David C. Gregory - 15 September 2005
//
//      Last modified:
//
//-----

class Gaussian
{
public:
    //Data Members
    double mean;
    double sigma;
    //Constructors and Destructors
    Gaussian(void);
    Gaussian(double meanIn, double sigmaIn);
    ~Gaussian(void);
    //Standard methods
    void Display ( );
    //Overloaded Class Operators
    Gaussian operator+(Gaussian &arg2);
    Gaussian operator+(double &arg2);
    Gaussian operator-(Gaussian &arg2);
    Gaussian operator-(double &arg2);
};

//Overloaded Non-Class Operators
Gaussian operator+(double &arg1, Gaussian &arg2);
Gaussian operator-(double &arg1, Gaussian &arg2);

```

APPENDIX E
UNCERTAINTY CLASS C++ SOURCE CODE

Gaussian.cpp

```
//-----
// NAVSEA Warfare Center - Dahlgren
//-----
//      Created by:      David C. Gregory - 15 September 2005
//
//      Last modified:
//
//-----

#include <iostream>
#include <math.h>
#include "gaussian.h"

using namespace std;

Gaussian::Gaussian(void)
{
}

Gaussian::Gaussian(double meanIn, double sigmaIn)
{
    mean = meanIn;
    sigma = sigmaIn;
}

Gaussian::~Gaussian(void)
{
}

void Gaussian::Display ( )
{
    cout << mean << ", " << sigma << endl;
}

Gaussian Gaussian::operator+ (Gaussian &arg2)
{
    Gaussian out;
    out.mean = mean + arg2.mean;
    out.sigma = sqrt(pow(sigma,2) + pow(arg2.sigma,2));
    return ( out );
}

Gaussian Gaussian::operator+ (double &arg2)
{
    Gaussian out;
    out.mean = mean + arg2;
    out.sigma = sigma;
    return ( out );
}

Gaussian operator+ (double &arg1, Gaussian &arg2)
{
    Gaussian out;
    out.mean = arg2.mean + arg1;
    out.sigma = arg2.sigma;
    return ( out );
}

Gaussian Gaussian::operator- (Gaussian &arg2)
{
    Gaussian out;
    out.mean = mean - arg2.mean;
    out.sigma = sqrt(pow(sigma,2) + pow(arg2.sigma,2));
    return ( out );
}

Gaussian Gaussian::operator- (double &arg2)
```

```
{  
    Gaussian out;  
    out.mean = mean - arg2;  
    out.sigma = sigma;  
    return ( out );  
}  
  
Gaussian operator- (double &arg1, Gaussian &arg2)  
{  
    Gaussian out;  
    out.mean = arg2.mean - arg1;  
    out.sigma = arg2.sigma;  
    return ( out );  
}
```

DISTRIBUTION

	<u>Copies (CD)</u>		<u>Copies (CD)</u>
DOD ACTIVITIES (CONUS)		INTERNAL	
ATTN CODE A76	1	B60 TECHNICAL LIBRARY	3
(TECHNICAL LIBRARY)		B	1
COMMANDING OFFICER		B50	1
CSSDD NSWC		B55	1
6703 W HIGHWAY 98		G24 (GREGORY)	1
PANAMA CITY FL 32407-7001			
 DEFENSE TECH INFORMATION CTR			
8725 JOHN J KINGMAN RD			
SUITE 0944			
FORT BELVOIR VA 22060-6218	2		
 NON-DOD ACTIVITIES (CONUS)			
ATTN JOHN CHIN	1		
GOVERNMENT DOCUMENTS SECTION			
LIBRARY OF CONGRESS			
101 INDEPENDENCE AVENUE SE			
WASHINGTON DC 20540-4172			
 THE CNA CORPORATION			
4825 MARK CENTER DRIVE			
ALEXANDRIA VA 22311-1850	1		

